2000

A Simple Method for Extracting Models from Protocol Code

David Lie Andy Chou Dawson Engler David L. Dill
Computer Systems Laboratory
Stanford University
Stanford CA 94305
E-mail: {davidlie,acc,engler,dill}@stanford.edu

Abstract

The use of model checking for validation requires that models of the underlying system be created. Creating such models is both difficult and error prone and as a result, verification is rarely used despite its advantages. In this paper, we present a method for automatically extracting models from low level software implementations. Our method is based on the use of an extensible compiler system, xg++, *to perform the extraction. The extracted model is combined with a model of the hardware, a description of correctness, and an initial state. The whole model is then checked with the* Murφ *model checker. As a case study, we apply our method to the cache coherence protocols of the Stanford FLASH multiprocessor. Our system has a number of advantages. First, it reduces the cost of creating models, which allows model checking to be used more frequently. Second, it increases the effectiveness of model checking since the automatically extracted models are more accurate and faithful to the underlying implementation. We found a total of 8 errors using our system. Two errors were global resource errors, which would be difficult to find through any other means. We feel the approach is applicable to other low level systems.*

1 Introduction

Our ability to design and manufacture increasingly complex systems is quickly outstripping our ability to verify those systems. The traditional method of verification is testing through trials. However, it becomes exponentially more difficult to fully exercise a system through testing as the number of control paths and corner cases increases. The result is increased system cost and decreased system reliability.

Formal verification methods are an attempt to solve this problem [17, 18, 21]. One option is model checking, which is the systematic and exhaustive exploration of the system state space. The computational complexity of model checking makes it impractical for full system models, so it is common to *abstract* system behavior (which means to suppress implementation details) or to *scale the system down* (which means to model a small instance of the system, say, three processors instead of 64). Performing one or both of these usually covers a greater range of behavior than conventional testing, and so uncovers bugs that testing does not. It is important to note that when used in this way, model checking abandons the traditional goal of formal verification, which is proving the "correctness" of a system, in favor of the more pragmatic goal of discovering bugs.

The difficulty of abstracting the design, a process that involves a great deal of manual effort, hampers the use of model checking in actual system design. Moreover, human errors in the manual abstraction result in missing bugs and causing false alarms during the verification process, further increasing the cost and reducing the usefulness of model checking. Such errors can be introduced both when constructing the model and as a result of "drift" as the actual system evolves [8].

This paper focuses on making model checking practical by developing techniques to automatically extract model descriptions from code. As a case study, we apply our methods to the cache coherence protocols used on the Stanford FLASH multiprocessor [15]. A FLASH protocol implementation consists of a collection of event-driven software *handlers* that are dispatched according to the requests that arrive on the various interfaces. These handlers, which run on the node controller, send messages on the I/O, processor, and network interfaces to maintain a directory of cache line states and service cache line requests.

Conventional simulation-based verification of FLASH has found many protocol bugs. Nevertheless, no protocol has booted perfectly on the hardware on the first try [7]. Using simulation to verify the protocols has been inadequate because of the limited and fixed detail level of the simulator and the high cost of simulating a large number of paths.

Though our approach could have been applied to a wide range of systems, FLASH protocol code has a number of features that make it a good case study of our approach. First, the catastrophic nature of coherence code bugs has already led others to use manually driven model checking to check FLASH protocols [19]. Thus, we can compare our method with a more conventional verification technique. Second, FLASH is representative of low level code that exists on a variety of embedded systems. It is highly optimized and difficult to read, and thus difficult to specify correctly. Finally, for the purpose of finding errors, FLASH represents a hard test: it is real, working, systems code that has undergone years of testing under simulation, on a real machine, and via formal verification. The main protocol we check, `dyn-ptr`, has been under constant use for over five years and has formed the basis for almost all experimental results on the hardware [13].

The critical enabling technology for our approach is an extensible compiler, *xg*++ [7, 10]. *xg*++ allows users to easily write domain-specific analysis extensions using a language called *metal*. There are two types of extensions: extensions that perform extraction, and extensions that perform translation. Extraction extensions select sections of protocol code to be modeled, while printing extensions translate the extracted protocol code into a *Mur*φ model description. *xg*++ uses program slicing to extract the selected sections of the implementation, while the translation is performed on the sliced-out abstract syntax tree (AST) [23]. Because the extraction is flexible, the author of the extensions can use human judgment to abstract away implementation details in order to focus on the important aspects and exploit all of the programming conventions used in the protocol code to do the best possible extraction. The use of *xg*++ for this application makes it feasible to write several customized translators to produce different models of the same underlying system, each focusing on different functionality. Each extracted model is then combined with a manually constructed model of the rest of the system, a correctness definition, and an initial state to form a complete model, which is verified using the *Mur*φ model checker.

Our main results are:

1. The approach is effective. We found eight hard errors in the code. All of these could have crashed the system. Two are errors that only occur on very specific sequences of events, which would make them difficult to find through testing.

2. The approach is practical. Our extraction and translation extensions are about 100 lines of code, which extract descriptions that are approximately 1000 lines from implementations that are about 10K lines. We did not have to make any modifications to the FLASH source, except to preprocessing macros.

3. The approach is more effective than manual verification – it found more bugs (the manual effort found none) and is significantly easier. Its increased effectiveness is largely due to the automatic extraction, which is more faithful to the implementation and checks many more features than the manually constructed model.

We are not claiming that these techniques are fully automatic. Rather, they automatically extract models from parts of the system whose implementations are understandable by *xg*++. For example, the FLASH network had to be manually modeled because it did not have an implementation that could be automatically processed.

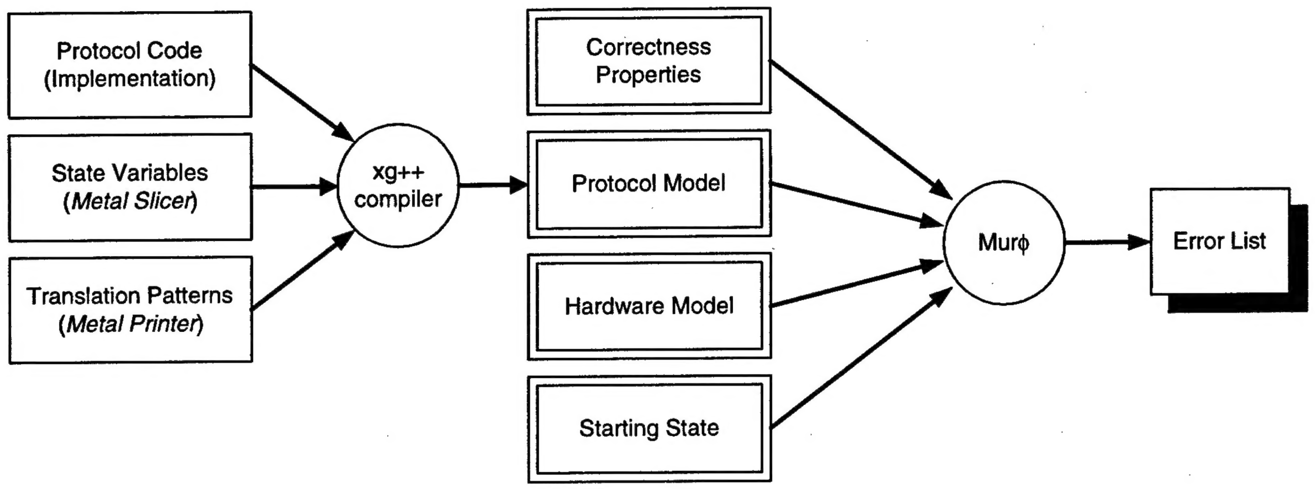
In this paper, we will explain our methodology and show how it was applied to the FLASH cache coherence protocols. We begin with a high level overview of how the system works in Section 2. The steps that require manual intervention are then detailed in Sections 3, 4, and 5. Section 6 presents the results of our verification of the FLASH protocols. We follow this by examining the accuracy of a manually constructed model of a FLASH protocol in Section 7. A comparison of our method to other similar methods is given in Section 8. Finally, we conclude the paper in Section 9.

2 Overview of the Extraction Method

In this section, we explain at a high level how our system works and then give an example of how an extracted model compares with a manually built model, as well as with the corresponding implementation code. Figure 1 illustrates the process of extracting and verifying models of the FLASH protocols. In our system, clients use the *xg*++ extension language, *metal*, to write the *metal slicer* extension, which specifies the state variables and subroutines that should be extracted into the model. The user also specifies rules in the *metal printer* extension for translating the sliced actions into a *Mur*φ model description. The *xg*++ compiler then takes these two *metal* extensions along with the original implementation code and produces a *Mur*φ model of the protocol.

Murφ is a model checker that uses explicit state enumeration with a Pascal-like language for specifying models. Model checkers perform verification by exhaustively searching the reachable states of a system for violations of user-specified invariants. In any given state, *Mur*φ will "nondeterministically" execute all possible outcomes. Each outcome is a new state, which is checked for correctness and then inserted into a table that contains visited states. If the state has been visited earlier, it is pruned and its successors are not visited again.

Before the *Mur*φ model checker can be applied, the protocol model must be combined with a model of the hardware on which the protocol runs. Unfortunately, there is



Protocol Code (Implementation)
State Variables (*Metal Slicer*)
Translation Patterns (*Metal Printer*)
xg++ compiler
Correctness Properties
Protocol Model
Hardware Model
Starting State
Murϕ
Error List

Figure 1. Flow chart of model extraction and verification

no easy way to automatically create this model since the hardware model must describe everything from the behavior of the processor interconnect to the functional units on the node controller. As a result, the user must still write the hardware model manually. The user must also specify a definition of correctness in the form of invariants and assertions, as well as a starting state. In the case of the FLASH protocols, these had to be manually specified since none could be extracted from the implementation.

At a high level, the user performs the following five steps:

1. Define the protocol state to be modeled. This is essentially a list of variables and functions relevant to the properties to be checked and comprises the *metal slicer* extension.

2. Add routines that insert or rewrite code. This step may add correctness checks or abstract away detail for the model. This comprises the *metal printer* extension.

3. Create a model of the hardware, correctness properties, and initial state. This process is entirely manual.

4. Combine the extracted model with the manually specified components to create a complete model that *Murφ* can check. This can be automated – in our case, a set of scripts performed this function.

5. Check the model with *Murφ*. The model checker provides an error trace if any correctness properties are violated.

However, these steps are not as difficult as they might appear. The first three steps are only necessary when the model is first defined, if there is a significant reimplementation, or if the scope of the verification effort changes. Since the *metal* extensions are applied to all handlers, they are independent of the number of handlers in the protocol code or the length of the code. The first two steps can be used to define several different protocol models, but these models will usually have nearly identical hardware models, correctness properties and starting states. Furthermore, though the last two steps might need to be performed more frequently than the first three, they are almost completely automated in our system, thus minimizing the incremental cost of keeping the model up to date if the underlying implementation changes.

Now, let us examine how this extraction method works on an actual segment of FLASH protocol code and how the extracted model compares with a manually specified model. Figure 3 shows a manually specified model of the FLASH protocol code in Figure 2, both with line number annotations that illustrate the correspondence between the FLASH implementation and model description. What the segment code actually does is not important for this example. Rather, the reader should notice that the paths of execution are dependent on the `hl` structure, which is the directory state. In addition, there are various `SEND` commands that cause messages to be sent on the processor (`PI`) or network (`NI`) interfaces. Finally, the reader should note that there are debugging assertions in the code that function just as assertions do in any C code. A *Murφ* model description consists of a series of rules. The rule bodies are executed when the rule precondition (the part before the ==>) is true. The structure of the model differs from the code because the author of the model chose to use two separate rules with different preconditions that are explicit "if" statements in the code. Despite this superficial difference, there is a clear mapping between statements in the *Murφ* description and the FLASH implementation.

The core observation motivating our work is that the cor-

```
void PILocalGet(void) {
  /* ... Boilerplate setup code ... */
  headLinkAddr =
    FAST_ADDRESS_TO_HEADLINKADDR(addr);
  FLDEBUG('h', "%u: headLinkAddr = %llx",
        procNum, headLinkAddr);
  READ_HEADLINK(headLinkAddr);
  nh.len = LEN_CACHELINE;
2,10 if (!hl.Pending) {
11      if (!hl.Dirty) {
        /* ... 37 lines deleted ... */
        ASSERT(!hl.IO);
        // The commented out ASSERT is
        // true 99.99% of the time, but is
        // not always
12!     // ASSERT(hl.Local);
        /*... deleted 15 lines ... */
14      PI_SEND(F_DATA, F_FREE, F_SWAP,
                F_NOWAIT, F_DEC, 1);
13      hl.Local = 1;
        /* ... deleted 14 lines */
3    } else {
5       ASSERT(!hl.List);
5       ASSERT(!hl.RealPtrs);
        FLSTAT_INC(procNum, readsCancelled);

        if (!hl.IO) {
5          ASSERT(hl.HeadPtr);
4          ASSERT(!hl.Local);
           nh.len = LEN_NODATA;
           /* setting opcode for send */
8          nh.msgType = MSG_GET;
           /* setting destination to
              node that called us */
8          nh.dest = hl.Ptr;
8          NI_SEND(THIRD, F_NODATA, F_FREE,
                   F_NOSWAP, F_NOWAIT, 12);
           /* ... deleted 12 lines ... */
6          hl.Pending = 1;
        }
```

Figure 2. Code associated with model description in Figure 3.

```
  Rule "PI Local Get (Else)"
1:  Cache.State = Invalid  & ! Cache.Wait
2:   & ! DH.Pending -- if pending NAK
3:   & DH.Dirty ==>
  Begin
4:  Assert !DH.Local "PILocalGet:L = A0";
5:  Assert DH.Head & !DH.List & DH.Real=0
                "PILocalGet:case D=1";
6:  DH.Pending := true;
7:  Cache.Wait := true;
8:  Send_Request(Home, DH.HPtr, Get,
                 Home, void);
  End;
  Rule "PI Local Get (Put)"
9:  Cache.State = Invalid & ! Cache.Wait
10:  & ! DH.Pending  -- if pending NAK
11:  & ! DH.Dirty ==>
  Begin
12:  Assert !DH.Local "PILocalGet:L = A0";
13:  DH.Local := true;
14:  CC_Put(Home, Memory);
  EndRule;
```

Figure 3. Partial *Murφ* model description for the `PILocalGet` handler in Figure 2.

```
Rule "PI Local Get"
  Cache.State = Invalid  & !
  Cache.Wait & Qspace(1) ==>
Begin
* HG_header.nh.len := len_data;
  if(hl.Pending = 0) then
    if(hl.Dirty = 0) then
      mbResult :=
        pi_send_func(procNum, nh);
      Local := 1;
    else
      assert(((hl.List = 0) != 0));
      assert(((hl.RealPtrs = 0) != 0));
      if(hl.IO = 0) then
        assert((hl.HeadPtr != 0));
        assert(((hl.Local = 0) != 0));
*       nh.len := len_nodata;
        nh.msgType := MSG_GET;
        nh.dest := hl.Ptr;
*       assert(nh.len = len_nodata);
        ni_send(2, 0, procNum, nh);
        hl.Pending := 1;
      else
      /* ... deleted ... */
```

Figure 4. An automatically extracted model of the FLASH code in Figure 2.

respondence between the model and the implementation is so strong that it should be possible to automatically build the model description from the code. We see this in Figure 4, which shows an automatically extracted model of the code in Figure 2 that was derived by our system. The *metal slicer* used to generate this description specifies that the `hl` and `nh` variables, the SEND functions, and the assertions should be extracted. The extracted model mirrors the code more closely than the manually constructed model and is richer in its description. Specifically, the header length assignments and assertions present in the code, which are marked with asterisks in the figure, are included. Automatic extraction makes it easy to model such additional features.

There are some differences between the manually constructed model and the extracted model. An example of this is line 7 in Figure 3, which does not appear in the protocol code because the hardware sets the cache state. When manually modeling a system, the user is free to mix actions of both the code and the hardware in the model. Our extracted model does not include hardware actions, which must be modeled manually instead. Another good example of both of these problems is line 12 in Figure 3. It is an assertion that has been removed from the implementation, but remains in the *Murφ* description. On the other hand, it did not cause any false positives because of translation mistakes elsewhere in the model. The problem of drift and translation mistakes between manually written models and the underlying implementations will be given more detailed treatment in Section 7.

Automatic extraction has two important benefits. First, the time required to create a model is reduced, and thus the user can specify a large number of small models that check orthogonal aspects of the same implementation. These small models make model checking computationally feasible, but do not sacrifice model detail. The other benefit is that the automatic extraction ensures that the extracted model is faithful to the original implementation.

3 The *Metal Slicer*

We now discuss how one uses *xg++* to extract a protocol model. *xg++* breaks the extraction down into two tasks. First, it uses the *metal slicer* to remove lines of code that do not affect the protocol state the user is interested in modeling, thus slicing the implementation down into a simpler model. Second, it translates the actions in the protocol implementation into abstracted actions in the model with the *metal printer*. We examine the *metal slicer* facility in this section and leave the *metal printer* for the next section.

The *metal slicer* allows the user to match arbitrary expression patterns in the implementation code to select a slice. Figure 5 gives a partial example of a *metal slicer* that extracts actions needed to check that the protocol code sets the length field of a packet header correctly. Each pattern declaration (`pat`) selects a portion of the FLASH code that will be extracted. For example, `pat length` indicates that the length field in the message header is to be extracted as part of the model description. Message sends and some directory values are also included: the former since the protocol must ensure that messages have their header length fields set correctly before sending, the latter so that paths dependent on the directory state are executed. In total, the *metal* extension of the `dyn-ptr` protocol is very compact, encompassing approximately 40 lines without comments.

Using a slicing algorithm [22, 23] automatically derives all code that affects the parts selected by the *metal* extensions. Our *xg++* based slicer computes a *backward slice* at the level of statements with an algorithm based on the program dependence graph (PDG) [12]. The nodes of a PDG represent program statements and the arcs represent the control and data dependencies between statements. Control dependencies occur when a statement can affect whether another statement is executed. For example, the true and false branches of an "if" statement are control dependent upon its condition. Data dependencies, on the other hand, link definitions of variables to their uses. Intuitively, these dependencies capture the flow of information from value producers or mutators to their consumers. Data dependencies can be calculated using the well-known "reaching definitions" data flow algorithm [1]. The slicing algorithm itself is implemented as a simple graph traversal of the PDG.

Standard slicing techniques have difficulty producing accurate slices in the presence of common C constructs such as pointers, unions, and unstructured control flow. However, since FLASH protocol code shares features common with low level systems code, it those troublesome features of C in very limited ways. As our slicer is configurable, we can have it automatically abstract those features by rewriting those sections, as will be demonstrated in Section 4.

By having an extensible compiler such as *xg++* the user need not understand the details of manipulating the compiler's internal data structures, nor does the user need to understand the implementation of program slicing algorithms.

4 The *Metal Printer*

Translation of the sliced code is accomplished by the *metal printer* extension, which allows the user to arbitrarily insert or rewrite actions in the model description. This facility allows the user to exploit domain-specific knowledge to create an optimal extraction. This capability is implemented by matching user-specified patterns against the abstract syntax tree as the slice is emitted. A pattern is enclosed by the first set of braces before the ==>. If the pattern matches, then the default output is suppressed and the pattern's action (after the ==> token) is executed to

```
sm len slicer {
  /* wildcard variables for pattern matching */
  decl { scalar } type, data, keep, swp, wait, nl;

  /* Pattern that will match all uses of the length field. */
  pat length = { nh.len };

  /* Patterns to match network and processor message
     sends, which use the length field. */
  pat sends =
      { NI_SEND(type, data, keep, swp, wait, nl) }
    | { PI_SEND(type, data, keep, swp, wait, nl) }
    ;
  /* Patterns to match accesses to directory entries */
  pat entries = { hl.Local } | { hl.Dirty } | { hl.List };

 /* Mark all matched patterns: the slicer will
    extract these and all code that influence them. */
 all: length | sends | entries ==> { mgk_tag(mgk_s); }
    ;
}
```

Figure 5. A *metal slicer* extension used to extract a model for verification of length field handling.

output user-specified code. The special emitter function `mgk_e` takes a `printf`-like format string augmented with `%t`, which allows matched pattern subtrees to be output as strings. One use of this facility is to include additional code that checks for correctness properties. This can help users tighten the verification of their models. Figure 6 gives an example of a protocol-specific printer that automatically inserts assertions before each `NI_SEND` to check that the length field in a network packet is correctly set before the packet is sent.

In addition to strengthening the correctness properties, the *metal printer* facility can be used to abstract away implementation details by taking advantage of FLASH domain-specific knowledge. There are three main areas in the FLASH verification where this is done: the emulation of bit operations in *Murφ*, reconstructing implicit types from C unions, and abstracting C data structures to reduce the state space.

Murφ is a more minimal language than C, and as such, does not provide some of the facilities that C does. Some examples of this are the bit operations that are often found in embedded systems code such as the FLASH protocols. These operations must be emulated by the *Murφ* model to allow for proper protocol modeling. We use our configurable printer to match uses of unsupported operations and rewrite them to call subroutines in the hardware model that emulate those actions.

Another complication results from the loose typing of C. In the `bitvector` protocol, the `Vector` variable represents a single node ID when there is only one sharer, but holds a bitvector of sharers when there is more than one. The `bitvector` protocol keeps the number of sharers in a separate variable. *Murφ* does not have enough type information to interpret these values since the union is implicit. However, we may leverage the *xg++* compiler to infer the type and rewrite the extracted output. Thus, in the extracted model, two variables replace the single `Vector` variable, one of which is a node ID, and the other, a list of nodes. Each access to `Vector` is replaced by an access to the appropriate variable while each modification becomes two modifications, one to each of the extracted variables.

Finally, to make model checking tractable, measures must be taken to limit the number of states. When manually constructing a model, the user will abstract data types when it is safe to do so. The same can be done with the automatic extraction if the compiler can be made to recognize instances where such abstractions can be made. For example, the `dyn-ptr` protocol uses a linked list to keep track of the sharers on a cache line. Implementing a literal linked list produces artificial state space explosion and complicates the model description since *Murφ* has no concept of pointers. Abstracting the linked list to an array makes the model much simpler and more efficient. The `dyn-ptr` protocol code manipulates the linked list through a set of functions, so every access is explicit. The linked list in the implementation is thus transformed into an array in the extracted model by configuring the *metal printer* to rewrite calls to the linked list accessing functions.

Interestingly, the extraction itself does not contribute directly to state explosion in the model checker. The size

```
sm printer tagged_printer {
decl { scalar } data, keep, swap, wait, dec, null, type;

all:
   /* Automatically insert length assertions before each send. */
  { NI_SEND(type, data, keep, swap, wait, null); } ==>
      {
         if (mgk_int_cst(data) != 0)
             mgk_e("assert(nh.len = len_data);");
         else
             mgk_e("assert(nh.len = len_nodata);");
         mgk_e("ni_send(%t, %t, procNum, nh);", type, swap);
      }
   /* rewrite 'len_cacheline' and 'len_word' as 'len_data' */
   | { len_cacheline } | { len_word } ==> { mgk_e("len_data"); }
   ;
```

Figure 6. A *metal printer* **extension used to insert length field assertions.**

of the state space and encoding is determined by the way the user chooses to specify the data structures in the model. Since the actions being extracted are executed atomically, redundancies and extra local variables in the extraction do not add extra states. They may result in additional computation time, but for large models this is usually not the limiting factor.

The *metal printer* is a good example of the benefit of having an extension-based system rather than an annotation-based one, since the annotation is effectively automated by the rules set in the printer. This alleviates the need for the user to manually place annotations throughout the code.

5 The Hardware Model, Correctness Definition, and Starting State

Before the model checker can be applied, it must be combined with a model of the hardware on which the protocol runs. In addition, a definition of correctness in the form of invariants and assertions must be specified, as well as a starting state for the model. The user verifying the system must manually create these components. Fortunately, these components usually do not change much in the course of system development.

In manually modeled systems, actions performed by the hardware and protocol can be interleaved. Because part of the modeling is done automatically in our system, this is no longer true. Rather, the hardware must be described separately so that it accurately models the interaction between hardware and the extracted model description of the protocol. There are two types of interaction that concern us. First, the protocol code can make calls to hardware functional units. Examples of this are the SEND instructions, which in reality are assembler instructions that cause the FLASH node controller to transmit messages. The other type of interaction is where the hardware causes certain parts of the protocol code to execute. For example, when the FLASH node controller receives a request, it consults a table that causes it to execute a specified piece of code called a *handler*.

On FLASH, the protocol code activates hardware units to perform functions. An example of this is the node controller logic that sends protocol messages out on the various I/O subsystem, processor, and network interfaces. The SEND instructions in the protocol code normally map to assembler commands, which are decoded and executed, eventually activating the interface logic. The hardware model maps these instructions to a subroutine that manipulates the model network and node controller data structures in the appropriate way to mimic this behavior. Another example is the "software queue", which is provided by the FLASH node controller hardware, where protocol handlers can suspend themselves in instances when there are not enough physical resources for them to complete their tasks, to be reactivated at a later time. Similarly, the hardware model has subroutines that act on data structures that mimic this queue. The instructions that the protocol code uses to activate the software queue are mapped onto these subroutines that we have provided.

Naturally, the hardware reactivates the suspended handlers at a later time. Thus, we arrive at the other form of interaction, where the hardware causes certain parts of the protocol code to run. The FLASH node controller has four input queues that can cause handlers to execute. One of these is the software queue where handlers are suspended. The others are input queues from the I/O, processor, and network interfaces. Before suspending themselves on the software queue, handlers store a continuation PC to a field in the software queue entries. If there is a valid entry present on the software queue, the node controller can select this

Invariants	Dynptr	BitV	RAC	Coma
The `RealPtrs` counter does not overflow (`RealPtrs` maintains the number of sharers)	X	X	X	X
Only a single master copy of each cache line exists (basic coherence)	X	X	X	X
A node can never put itself on the sharing list (sharing list is only for remote nodes)	X	X	X	X
No outstanding requests on cache lines that are already in `Exclusive` state	X	X	X	X
Nodes do not send network messages to themselves	X	X	X	X
Nodes never overflow their network queues	X	X	X	X
Nodes never overflow their software queues (queue used to suspend handlers)	X	X	X	X
The protocol never tries to invalidate an exclusive line	X	X	X	X
Protocol can only put data into the processor's cache in response to a request	X	X	X	X
All processor message header opcode fields are set to valid opcodes	X	X	X	X
Opcode XOR operations always occur on known opcodes (invalid opcodes are never created)	X	X	X	X
If there is no sharer in the `HeadPtr`, the sharing list is empty	X		X	X
If the sharing list is not empty, `RealPtrs`, the number of sharers is greater than zero	X		X	X
The protocol state is pending while waiting for invalidations	X		X	X
When a line is dirty, the sharing list is empty (this is only true for if there are no handler suspensions)		X		

Table 1. Description of all invariants checked.

suspended handler to be serviced by jumping to the continuation PC. An enumerated variable whose values represent all the possible entry points that the continuation PC's can take together with a dispatch function that mimics the hardware jump mechanism models this behavior. The dispatch mechanism for the other three queues is similar. Each message that arrives on one of the I/O, processor, or network interfaces, contains an opcode that indicates the message type. A JumpTable configuration file that indicates which handler is executed depending on the type of message that arrives, is used to program the hardware dispatch. These dispatch conditions can be easily transformed into the preconditions that guard each extracted handler rule. In fact, the mapping is simple enough that in our FLASH verification, this process was automated with a simple script.

In addition to the hardware model, a correctness definition must be provided. Table 1 gives a list of invariants that we check. Some of these are invariants about the modeled hardware components. For example, a node cannot send a packet to itself – the network will not route such a request properly. On the other hand, some invariants are model specific. For example, in the `dyn-ptr` protocol, if the list of sharers is non-empty, then the head pointer cannot be NULL. The `bitvector` protocol does not use a linked list so this invariant cannot be applied to that protocol. In addition, the invariants may change depending on what aspects of the protocol are modeled. The ability to specify protocol specific invariants allows the user to provide very specific correctness conditions. However, as we see here, out of 15 invariants, 11 apply to all cases. Thus, in our FLASH verification, the invariants are largely independent of the protocol model.

Finally, a starting state must be provided. For FLASH, this is the state of the machine at power-on, meaning that all valid memory is at its home node and the directory entries are all blank.

6 Results

With an extracted *Murφ* model, we found a total of eight bugs in two of the four FLASH protocols modeled. We found six errors in `dyn-ptr` (four network header bugs, two counter overflows) and two in `bitvector`. In contrast, the manual verification of `dyn-ptr` found no bugs.

The results of the verification are given in Table 2. The size of the manually built component, which includes the hardware model, invariants, and starting state, changes slightly between protocols because of the different invariants and needs of each model. Note that the automatic extraction reduces the number of manually written lines by a factor of two or more. What is even more significant is that since our method faithfully extracts a model, the user need not understand every detail of the protocols to produce one.

The automatically inserted assertions described in Sec-

Protocol (Max Processors)	Errors found	Protocol Size (lines)	Extracted Model (lines)	Manual Model (lines)	*Metal* Size (lines)
Dyn-Ptr(n=4)	6	12K	1100	1000	99
Bitvector(n=4)	2	8K	700	1000	100
RAC(n=4)	0	10K	1500	1200	119
Coma(n=4)	0	15K	2800	1400	159

Table 2. The results of verifying four protocols.

tion 2 found four bugs in `dyn-ptr`. To improve performance, the protocol speculatively sets the field to optimize for the common case. The extractor was able to determine what kind of message the protocol was sending and assertions were automatically placed before each send operation to ensure that the data length field was set correctly. Because *Murφ* exhaustively exercises all paths, it detected the four uncommon cases where the speculation was false, but there was no correction code.

After fixing the preceding bugs, two subtle counter overflow errors were found. Both errors involved miscalculations of the maximum number of sharers that a counter had to record. They are particularly malicious in that they only manifest themselves as a result of a single rare interleaving of events.

The first bug involves a performance optimization, *limit search*, used in the `dyn-ptr` protocol. The problem with using a linked list, as `dyn-ptr` does, is that the worst case overhead of searching for a single sharer becomes linear with the number of sharers. Such a search occurs when a node n is already on the list and evicts the cache line due to a capacity or conflict cache miss. As a result, n should no longer be on the sharing list and needs to be removed. In practice, a linked list traversal on every cache line eviction is far too costly. However, the sharer must be removed from the list or repeated evictions and requests can cause the list to grow without bound. The limit search optimization makes the cost of cache line eviction independent of list length. If the sharer that is to be removed is not found after searching a fixed number of list entries (the limit), a counter, `StalePtrs`, is incremented to indicate that there is a "stale" sharer in the list. When `StalePtrs` reaches its maximum value, all sharers on the list are invalidated to remove the duplicate sharers. A second counter `RealPtrs` is used to keep track of the list size. It is incremented on every sharer addition and decremented on every sharer deletion. As a result, `RealPtrs` must be large enough to hold the maximum number of sharers on a list, which is the maximum value of `StalePtrs` plus the number of nodes on the system[1].

Unfortunately, due to the reallocation of bits in the structure used to hold these counters, the size of `RealPtrs` was 7 bits while `StalePtrs` was 10 bits, causing `RealPtrs` to overflow on the actual machine. The model checker detects a clear sequence of events that leads to the to the counter overflow.

The second overflow error also occurred on the `RealPtrs` counter, which maintains a count of the number of sharers. In the absence of limit search (maximum value of `StalePtrs` is zero), the maximum value of `RealPtrs` was thought to be the maximum number of physical nodes that can be supported on a system. However, a specific interleaving of messages can result in a `RealPtrs` count of one greater than the number of nodes, thus breaking the rule. Because the implementation of the protocol has space allocated to `RealPtrs` for 128 nodes regardless of the number of nodes on the system, this bug never occurs, even after extensive use of the machine, because only 72 nodes exist and thus the `RealPtrs` limit is never tested. However, in the future if the width of `RealPtrs` decreases or a larger machine is built, this would cause failures.

Finally, there were two errors found in the `bitvector` protocol. We found one case where a message was sent on the wrong network lane. Neither the simulator nor the hardware checks that the messages are on the correct lanes, and there is no manually built model of the `bitvector` protocol that would have caught this error. A false assertion was also discovered in the `bitvector` protocol. It checked an incorrect invariant about the I/O state. It was not caught earlier because assertions are usually disabled on the hardware and I/O is not modeled in simulation.

7 Imposing Model Descriptions

We also studied the extent to which manually described models can be inaccurate either due to translation errors or "drift." While it is not clear which results in more errors, it is immaterial since the end result is the same – bugs may be missed if a model is specified incorrectly. We use *xg++* to create an automatic "checker" that looks for semantic differences between a model and the matching protocol code. To collect data, we apply this *xg++* extension to the model of the `dyn-ptr` protocol created by Park and Dill and the current FLASH protocol code [19]. This data will give us

[1]Actually, this is not really true because of the next bug.

an idea of how faithful manually written model descriptions are to their underlying implementations.

Rules in *Murφ* contain a precondition that guards actions. We extract each rule's actions and precondition using a modified version of the *Murφ* front-end parser. Included in this process is converting strongly typed *Murφ* variables to C's weak type system. To translate semantics from the model to the protocol code, we provide a table that maps each model variable to its FLASH equivalent. For each rule, the extension uses a heuristic on the name of the rule to determine the corresponding FLASH handler. A *xg++* extension uses this mapping to attempt to match the actions of each rule to those in its handler. It searches for a path in the FLASH handler that will satisfy the rule's preconditions by observing all conditionals, assignments, and assertions. For example, given the precondition `!DH.Pending & DH.Dirty`, it searches for a sequence that implies `DH.Pending` to be 0 and `DH.Dirty` to be 1. This can be inferred by tracking "if" statements, assertions, and assignments in the code. If no such mapping exists, the extension emits an error message.

After a path satisfying the precondition has been found, the extension attempts to match all actions associated with that rule along that path. The manual model description is simple enough that there are only four types of actions to find: assignments, assertions, decrements, and message sends. Assignments and decrements can be transliterated from *Murφ* to FLASH. Note that conditionals that check that the variable has the value assigned in the *Murφ* model also implied that the assignment is matched. This condition arises when the model omits details. Assertions in *Murφ* are simple binary boolean operations consisting of one operator (equal, not-equal) and two operands. They can be matched by either an assertion in the implementation or a conditional that implies that they are true. Message sends on the other hand require special treatment since their operations can be more diffuse in the FLASH code. For example, the message send at line 8 in Figure 3 encompasses three separate statements in Figure 2. For a send, the message opcode and outgoing lane arguments are checked as well as outgoing interface. Note that the extension only maps elements in the model onto elements in the protocol code.

Every action in the manual model description was checked against the implementation. This found 14 differences between the model and the implementation. These differences fall into four categories: semantically non-equivalent code rearrangement, hard errors in the translation of the model, semantically equivalent syntactic differences, and incidental differences resulting from modeling a subset of the implementation. We consider the first two categories to be translation errors that could hide potential bugs.

There were two cases in the first category. These consisted of cases where assertions that were guarded by "if" statements in the model had been hoisted past the corresponding "if" statements in the protocol code. Guarding the assertions with an extraneous conditional made the manual model description weaker than the implementation since the assertions are only checked on that path.

There were two errors in translation, which could mask errors in the model itself. In one, the model of the `NILocalGetXDelayed` handler first checks that is legal to assign the value 0 to the variable `DH.Real` before making the assignment. In the actual protocol implementation, `RealPtrs` is a counter for the number of sharers on the linked list. Thus, setting it to zero is a violation of the way this counter should have been used. The other error involves the assertion shown in line 12 of Figure 2 and Figure 3. This assertion is incorrect, but survived verification because the manual model description lacked the details to trigger it.

There were six cases where implementation code was translated to semantically equivalent but syntactically dissimilar model code. For example, in the `NIInvalAckDelayed`, the protocol decrements the counter `RealPtrs` and then tests for equality to `0`. The model tests if `RealPtrs` equals `1` and then decrements. Since handlers on the same node run sequentially, these actions are equivalent.

Finally, there were four incidental differences. In one, the model indicates that an `INVAL_ACK` message should be sent, but the implementation sends an `INVAL` message instead. In reality, the two opcodes have the same underlying bit encoding so they are equivalent even though they are syntactically different. Other examples arose because the model only partially describes the protocol, and so must make assumptions about the modeled state.

In summary, 14 differences were found: two rearrangements, one translation error that weakened the model, one false assertion that was hidden by a simplified hardware model, and ten incidental differences. These differences illustrate the problems caused by manual modeling both in its initial construction and in its modification to track implementation changes.

8 Related Work

In previous work, *xg++* was used to build a set of static checkers for both the FLASH protocols [7] and for general systems code [10]. This paper's use of model checking and slicing-based model construction is a fundamentally different approach to finding errors. The methods of both papers are largely complementary. The errors found in this paper require dynamic information and can catch very convoluted race conditions. In contrast, the static checkers are shallower, but more light-weight and do not need to simulate any protocol code.

We could have potentially used other open compilers to extract models. These include Lord's ctool [16], Crew's ASTLOG [9], Shigeru Chiba's Open C++ [5, 6], and Srivastava and Eustace's ATOM [20] object-code modification system. However, it appears that the first three would have required extensive retooling to support the analysis we needed. ATOM, on the other hand, works at too low a level for our purposes.

There is one published example of model checking being used on an *implementation* of a cache coherence protocol [11]. In this case, the implementation is in hardware. The model checking technique was to use refinement in Cadence SMV. So far as we know, no one else has been able to apply this verification approach.

There have only been a few systems to do *computer-assisted* model extraction. The Bandera system is a sophisticated model extractor for Java programs [8]. Bandera has two methods for extraction. The first is a program slicer that accepts temporal properties as slicing criteria and uses sophisticated static analysis algorithms to do accurate slicing. Effective slicing in Java requires new slicing algorithms for multi-threaded programs. The slicer removes irrelevant code and variables that could otherwise blow up the state space during model checking. The second technique is data abstraction. The user maps data types to a small set of abstract values. Abstract versions of operations applied to these data types are defined. Since the number of states visited by a model checker is a function of the number of distinct values each variable can have, this also has the potential for greatly reducing the state space during model checking.

Our approach is more pragmatic than Bandera's. Our method permits the use of an open-ended collection of *ad hoc* extraction methods, and is optimized for finding bugs. It would be difficult to imagine handling the FLASH protocol implementation without this flexibility. Bandera has not been successfully applied to examples comparable in complexity to the FLASH protocols.

The SLAM project at Microsoft Research extracts a program with only boolean variables from C code [2, 3]. These variables represent boolean conditions in the original code. This program is model checked, and the resulting counterexamples are verified using symbolic execution and decision procedures. If a counterexample is found to be a false alarm, constraints are added to the boolean program to improve the model. The goal of the project is to check assertions in the code. In contrast, we are extracting a model, then using *Murφ* to check higher-level properties of several instances of the models running concurrently. It is difficult to imagine solving this problem with SLAM because of limitations on the properties it can check and the scale of the model checking problem they would have.

An approach that is similar to ours in philosophy was used to check Lucent's PathStar system [14]. A "control skeleton," which consists of only the control constructs of a system, was extracted using a simplified C parser, and then selected constructs (such as message sends and receives) were extracted from the original source using a collection of pattern matching rules. The result was checked using the SPIN model checker, which is an explicit state model checker somewhat like *Murφ*.

An alternative approach to ours is the Teapot system, which is a programming environment for software implementations of multiprocessor cache coherence protocols [4]. Teapot couples a domain-specification language for writing cache coherence protocols with *Murφ*, which is used to verify the protocols. The protocols are automatically translated to C after verification. Program generation, as in Teapot, is a good approach when applicable. However, it relies on the availability of adequate compilation techniques for the highly specialized hardware used in the multiprocessor interconnect. There are numerous examples in the FLASH protocol where hand optimization was necessary because the compiler was inadequate. The customizable extraction methods described in this paper can be applied in the majority of cases when program generation is impractical.

9 Conclusion

We have demonstrated a simple approach to automatically extracting models from protocol code. Our method both reduces the effort of using model checking and makes it more effective by ensuring that the extracted model is more faithful to the original protocol code. We were able to apply model checking to four protocols in less time than it took to manually verify just one. One of the great benefits is that the amount of manual labor required is reduced by a significant amount. In addition, our models are more complete and found errors that eluded the manual verification process. The automatic nature of the extraction also reduces the problem of drift, ensuring that the model that is checked closely tracks the underlying implementation. In total, our method found eight protocol bugs that were not found by the manual verification. We also found four discrepancies between the manually described model and the implementation that may have accounted for some of the missed bugs. Our method, though automatic, does not impact the state space of the model created.

The core of our approach is the use of an extensible compiler. Compilers understand code at a programming language level. We leverage this understanding to build a model from the implementation code. This is accomplished through two facilities provided to us by *xg++*. One is the *metal slicer*, which is used to select features in the implementation to extract. The other is the *metal printer*, which allows the user both to specify additional checks to tighten the criteria for correctness and to specify rules for recognizing opportunities to perform abstraction. In combination with a model checker that takes imperative language input such as *Murφ*, models can be quickly and easily constructed. The benefit here is that a greater amount of a system can be checked by extracting many orthogonal models and checking each separately. While the method is not fully automatic, some of the verification tasks which are both tedious and error prone have been automated.

We feel that this method is applicable to a range of problems encountered while debugging and verifying low level systems. It seems particularly effective on code found on embedded applications where the code is easily analyzed by tools but difficult for humans to read.

Acknowledgements

We thank Mark Heinrich and Vijayaraghavan (Ravi) Soundararajan for enlightening us on the mysteries of FLASH. We also thank Mark Horowitz for his help and guidance. Finally, we thank Diane Tang for contributing valuable advice on the writing of this paper. This research was supported in part by DARPA contract MDA904-98-C-A933, by SAL contract NAS1-98139, and by a grant from the Stanford Networking Research Center.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. of the SPIN 2000 Workshop on Model Checking of Software (LNCS 1885, Springer)*, pages 242–252, Aug. 2000.

[3] T. Ball and S. K. Rajamani. Checking temporal properties of software with boolean programs. In *Proc. of the Workshop on Advances in Verification (with CAV 2000)*, 2000.

[4] S. Chandra, B. Richards, and J. Larus. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering*, 25(3):317–33, May 1999.

[5] S. Chiba. Open C++ programmer's guide. Technical Report TR93-93-3, Dept. of information science, University of Tokyo, 1993.

[6] S. Chiba. A metaobject protocol for C++. In *Conf. Proc. Object-oriented Programming Systems, Languages, and Applications (OOPSLA95)*, pages 285–299, Oct. 1995.

[7] A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. In *Proc. of the Ninth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. of the Intl. Conf. On Software Engineering (ICSE 2000)*, pages 263–276, Nov. 2000.

[9] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proc. of the First Conf. on Domain Specific Languages*, pages 229–242, Oct. 1997.

[10] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation (OSDI 2000)*, pages 23–25, Sept. 2000.

[11] A. T. Eriksson and K. L. McMillan. Using formal verification analysis methods on the critical path in systems design: A case study. In *Proc. of the 7th Intl. Conf. on Computer Aided Verification (CAV95)*, pages 367–380, July 1995.

[12] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

[13] M. Heinrich. *The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols.* PhD thesis, Stanford University, Oct. 1998.

[14] G. Holzmann and M. Smith. Software model checking: Extracting verification models from source code. In *Invited Paper in the Proc. PSTV/FORTE99 Publ. Kluwer*, 1999.

[15] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein R. Simoni, K. Gharachorloo, J. Chapin D. Nakahira, J. Baxter, M. H. A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proc. of the 21st Intl. Symp. on Computer Architecture*, Apr. 1994.

[16] T. Lord. Application specific static code checking for C programs: Ctool. In *~twaddle: A Digital Zine (version 1.0)*, 1997.

[17] K. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proc. of the Intl. Symp. on Shared Memory Multiprocessing (ISSMM91)*, pages 242–251. Tokyo, Japan Inf. Process. Soc., 1991.

[18] G. Nelson. *Techniques for program verification.* Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.

[19] S. Park and D. L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proc. of the 8th ACM Symp. on Parallel Algorithsm and Architectures*, pages 288–296, June 1996.

[20] A. Srivastava and A. Eustace. Atom - a system for building customized program analysis tools. In *Proc. of the SIGPLAN '94 Conf. on Programming Language Design and Implementation*, 1994.

[21] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conf. Proc.*, 1995.

[22] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[23] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.